Bachelor thesis
Computing Science

Radboud University

Accessible Process Mining:
Developing a Business User Application Utilizing the Heuristic Miner Algorithm

Author:
Sven van der Post
s1028679

First supervisor/assessor:
Dr. Y. Shapovalova
`yuliya.shapovalova@ru.nl`

Second supervisor:
Prof. dr. ir. D. Hiemstr
`hiemstra@cs.ru.nl`

Second assessor:
Prof. dr. T.M. Heskes
`t.heskes@science.ru.nl`

June 23, 2023

Abstract

Process Mining is an area of data science which is involved with discovering, validating and improving workflows. Log data generated from any type of process serves as an input for algorithms which yield models used to understand the process performance. By inspecting these models and drawing comparisons with the documented process flow insights can be gained about bottlenecks and other areas of improvement. There are a lot of opportunities to be gained by applying Process Mining to organisations but, with Process Mining being a relatively young discipline, the offer of software solutions is limited and often costly or made for academic purposes. To address this gap I developed a proof of concept Process Mining software application on behalf of VANL, a company specialized in public transport data and tailored manufacturing software. The application was written using an inhouse framework and at its core makes use of the Heuristic Miner algorithm. This algorithm infers a process model based on frequency and ordering patterns in the logging data. The resulting model is a Work Flow Net, a specific type of Petri net designed for providing accurate visual representations and execution properties of the process flow. Besides this descriptive model the application is also able to render two other Process Mining models, a Dotted Chart and a Flow Network. Both these models are less complex in nature and are often used to inspect and evaluate logging data in the data collection phase. When the desired functionality of the application was met the Work Flow Nets where evaluated on conformance to the dataset using the Partial Parsing Measure criteria. This unveiled the generated networks where not able to replicate the desired process behaviour and often contained mistakes leading to deadlocks. Despite the underwhelming performance of the Heuristic Miner algorithm the application as a whole still proved to provide value to the business user.

Contents

Chapter 1

Introduction

The real manufacturing process is often a challenging process to model [13]. Usually there exists a model or description of the *expected flow*, the planned route a product takes through the process. The *actual flow* however, the path a product takes in reality, can drastically deviate from this intended route [3]. These deviations can take different forms. Most of the times such a deviation is a secluded occurrence, a few products go missing and end up in the wrong place. The cause of such cases is often faulty machinery, incorrect information or human error. Sometimes however, the problem is more fundamental. In such cases there is a structural problem in the flow which means that a significant amount of products diverge in the same pattern. Discovering both these kind of deviations is very important for a manufacturer since any reduction in efficiency can result in a loss of possible profit. Finding these deviations is not an easy task however. Traditionally this is done by hand, meaning manual analysis on the available data or sometimes even inspection of the factory hall. This is a tedious and time consuming task which has to be repeated for every single occurrence. In order to reduce the efforts needed, a family of techniques was developed which was labelled 'process mining'. These techniques all perform analysis on processes based on event logs and provide some sort of 'X-ray' of the business process, revealing what really goes on inside.[13].

This thesis presents an implementation of a Process Mining application which is focused on the business user. The application will be able to analyse the provided logs of a business and produce various models. The analyst of said business can then use the models to require insights of the business process. This will drastically reduce the time and effort needed for process analysis and give the managing parties information on how to improve the process. Currently there are existing applications which offer process mining analysis, both free to use and commercial. The difference between these applications and the one presented in this thesis is that the former are made to be used by people schooled in the knowledge area of data mining. These workers are often scarce too costly for the more modest sized organizations as they need to be high schooled. The new implementation presented in this thesis provides a simpler interface and user experience so the application can also be used by process managers on the business side.

Chapter 2

Preliminaries

2.1 Process Mining

The current age we live in is often dubbed the *information age* , and with good reason [6]. In this digital era all sorts of data is stored and shared in abundance, with no less than 2.5 quintillion bytes being generated on a daily basis [9]. With digital data as a new resource countless scientific disciples have emerged with the purpose of transforming this matter into valuable information. Process mining is such a field, and focuses on turning the data gathered from executing a process, the activity of *logging*, into information *about* said process. The purpose is to provide insight into the actual functioning of a process and identify opportunities for improvement. Logging data is the most conventional data type used for process mining. This comprises recordings of any *activity* being carried out in a process. For instance, such an activity could involve wrapping a package in a delivery warehouse or clicking a certain button on a website. Each of these actions represents a step within a larger process, the *delivery* process or the *user* process, respectively. Such an individual activity conducted in the *flow* of a process is called an event. The flow itself, the sequence followed by a single entity through the process, is called a case. The collection of cases derived from a process is termed the log.
In this section we will give an overview of the key terminology employed in process mining and offer concise explanations and definitions for each term.

2.1.1 Event

An event ε is a tuple of the form (ID,Tag,TimeStart?,TimeStop?,info$_1$,...,info$_m$) and represents a single entry in a *Case*. Every event has an unique ID and is paired with a 'tag', a word describing the activity. Ideally this event also has a timestamp with the start end end time of the activity but these are optional, as long as the sequence of events is known. To provide additional context about an activity more information can be added to the event prop-

erties, such as the use of resources or cost of the operation.
We define π to be the projection function. $\pi_{activity}$ returns the tag of the event. Because the tuple can be of arbitrary length as much info can be added as desired, indexing is also possible for any index number $i \in \{1, \ldots, m\}$. π_2 also returns the tag.

2.1.2 Case

Each case $\mathcal{C}$ contains a trace σ of *Events.* The trace is ordered chronologically either by the shape of the log or by ordering it on the timestamp data of its *Events.* The series in a case regard a single 'line' in the process c.q. the lifecycle of a single instance. Every case has an unique ID, either generated or extracted from the logging data. We define the notation of a case as follows: $< \varepsilon_1, \varepsilon_2, \varepsilon_3 >$ without its ID or as $< 115 \mid \varepsilon_1, \varepsilon_2, \varepsilon_3 >$ with ID, where ε_n is the activity of event n in $\mathcal{C}$ with $n \in \{1, \ldots, |\mathcal{C}|\}$.

2.1.3 Log

A log W is a collection of *Cases.* This collection represents the dataset which is used as input for the Miner. Apart from the main set a Log has three more properties. The set $\mathcal{A}$ of unique activities present in W and two *Events* defined as the *start* and *finish* of the log. These two events are necessary as they are requirements for a WF-net. This requirement also dictates that each *Case* in the log must thus begin with the *start* activity and end with the *finish* activity. For some cases this is innate in the data of the log, for other cases the log needs to be modified by prefixing the *start* activity and/or appending the *finish* activity. To clarify the structure of a log an example of a log with three different cases can be found in Table 2.1. Each event in the case has besides the required *tag* property also a *resource* property specifying the operator, and a *cost* property denoting the cost of the action.

2.2 Petri Nets

One of the models which we can extract from the log using process mining is a Petri Net. A Petri Net, developed by the work of Carl Adam Petri, is an abstract, formal model of information flow [11]. It can be described as a directed bipartite graph that has two types of nodes: places and transitions. They are most often depicted as white circles and rectangles, respectively. Edges connect a place to a transition or vice versa, but never to another node of the same kind. The state of a Petri Net is determined by the distribution of *tokens* over its places, also referred to as its *marking.* A place can contain any number of tokens, which are depicted by black dots inside the circle of

Case ID	Event ID	Properties Tag	Resource *(Optional)*	Cost *(Optional)*
1	10043	"Snijden"	Sven	10
	10044	"Sealen"	Ella	35
	10047	"Inspecteren"	Jesse	20
2	10048	"Snijden"	Han	10
	10051	"Sealen"	Ella	35
	10052	"Versturen"	Sven	10
3	10053	"Uitladen"	Abe	5
	10054	"Naaien"	Abe	15
	10056	"Versturen"	Ella	10

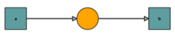
Table 2.1: Example of a log with three cases. Every case and event is paired with an id for identification. An event has a required *Tag* property but can contain multiple additional properties for context, such as the Resource used and its associated Cost

a place. A formalization of a Petri Net and its properties by Wil van der Aalst [8] can be found below.

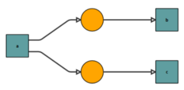
Definition 2.2.1. A Petri Net is a triplet $N = (P, T, F)$ where P is a finite set of places, T is a finite set of transitions such that $P \cap T = \emptyset$, and $F \subseteq (P \times T) \cup (T \times P)$ is a set of directed arcs, called the flow relation. A marked Petri Net is a pair (N, M) where $N = (P, T, F)$ is a Petri Net and where $M \in \mathbb{B}(P)$ is a multi-set over P denoting the marking of the net. The set of all marked Petri Nets is denoted $\mathcal{N}$.

For completeness we formalize a notation for the input and output of places or transitions. Elements of $P \cup T$ are called *nodes*. If there is a directed edge from a node x to another node y then, and only then, x is an *input node* of y ($(x, y) \in F$). In the opposite case, x is an *output node* of y ($(y, x) \in F$). This can be translated to set notation.
For any $x, y \in P \cup T, \bullet x = \{y \mid (y, x) \in F\}$ and $x\bullet = \{y \mid (x, y) \in F\}$
This operator can also be chained to create a union of sets.
For any $x \in P \cup T, \bullet\bullet x = \bigcup_{i=1}^{n} \{\bullet y \mid y \in \bullet x\}$ where $n = |\bullet x|$

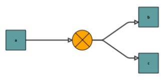
The structure of the network is static, but it also has dynamic properties that result from its execution. Its execution consists of a single rule, the firing rule. When each of the input places of a transition contain a token, the transition is *enabled*. Any enabled transition in the net can *fire* arbitrarily which will consume one token from each input place and produce one token for each output place. As a result of this process the amount of tokens in



(a) Sequence



(b) AND



(c) XOR

Figure 2.1: Some basic templates that can be modelled using Petri net notation. A transition is depicted as a blue square and a place as an orange circle. In a sequential template transition b can only fire after transition a. In the AND template after a both b and c can fire. The XOR template only permits either b or c to fire after a since it only produces 1 token in its shared output place.

the net can change. For example, lets consider a transition which has two marked input places and a single output place. When it fires it will consume the two tokens from the input and produce a single token in the output when fired, thus decreasing the total amount of tokens by one. The sum of tokens in the net can also increase in the opposite situation, when a transition has more output than input places.

Petri Nets have a strong theoretical basis and can capture concurrency well. Moreover, a wide range of powerful analysis techniques and tools exists [12]. With these semantics it is possible to model many different behaviours, the behaviours most important to process flows are the sequential, AND and XOR template. These templates can be found in Figure 2.1. The sequence template describes the causal dependency between two activities where activity A is required to fire before activity B can fire. The AND template represents the splitting of the process flow into two or more branches, where each branch gets executed concurrently. Once A is fired both B and C can fire, in no specific order. The XOR template serves as the mutual exclusion of two or more flows. After A is fired only B or C can fire, never both.

For process mining we use a subclass of Petri nets called Workflow nets (WF-net), the difference being that a WF-net must have a dedicated 'start' and 'finish' activity. Andrea Burattin characterizes a WF-net as follows [5].

Definition 2.2.2. A WF-net is a Petri net $N = (P, T, F)$ such that:

a. P contains exactly one place i with no incoming arcs (the starting point of the process);

b. P contains exactly one place o with no outgoing arcs (the finishing point of the process);

c. if we consider $\bar{t} \notin P \cup T$, and we use it to connect o and i (so to obtain the so called "short-circuited" net: $\overline{N} = (P, T \cup \{\bar{t}\}), F \cup \{(o, \bar{t}), (\bar{t}, i)\})$, the new net is strongly connected (i.e. there is a direct path between any pair of nodes).

Not every WF-net correctly represents a process, such an inaccurate WF-net could exhibit errors such as a deadlock, meaning there exists a reachable marking where no transition is enabled. Furthermore also errors such as activities that can never become active or tokens being left in the net after termination. For this reason we define the following correctness criterion [2] [1].

Definition 2.2.3. Let $N = (P, T, F)$ be a WF-net with input place i and output place o. N is *sound* if and only if

- (*safeness*) Places cannot hold multiple tokens at the same time;
- (*proper completion*) o is always in the set of reachable markings from i;
- (*option to complete*) For any marking M in the set of reachable markings from i, o is in the set of reachable markings from M;
- (*absence of dead parts*) For any $t \in T$, there is a firing sequence enabling t.

The soundness of a WF-net can be verified using analysis techniques.

A way to verify the fitness of a WF-net is by executing or 'parsing' the training data on the net. For each trace in the log (or a subset) a token will be placed in the starting place. Every event in the case will than be attempted to fire in the order it occurs in the trace. If no transition is unable to fire the trace is parsed successfully. The total number of successfully parsed traces (c) is then divided by the total amount of traces in the log (t) to form the **Parsing Measure** [7].

$$PM_w = \frac{c}{t}$$

Sometimes this measure is a bit too robust, as a single deadlock in the net just before the *finish* could easily result in a Parsing Measure of 0, therefore we can also consider each partially parsed trace. For the **Partial Parsing Measure** we keep track of the percentage of events which is correctly executed from a trace and sum these together(e). This sum is than again divided by the total amount of traces(t).

$$PPM_w = \frac{e}{t}$$

Chapter 3

Mined properties

These properties are computed during the mining phase, which will be explained in section 4.1.2. To make this section more comprehensible we will provide a brief explanation in this chapter.

The interpretation of a trace σ depends on whether the events are provided with a timestamp or not. When the events are stamped they inherently contain a property for the *start* and *finish* of the activity. When they are not we can artificially create these by setting each property to increasing values as to the order of the trace. The trace $\sigma = < a, b, c >$ will become its equal counterpart $\sigma' = < a_{start}, a_{finish}, b_{start}, b_{finish}, c_{start}, c_{finish} >$. By applying this transformation we can implement the same relation and measure definitions to both timestamped and ordered data.
Conversion from timed to ordered data is done by deleting the *finish* event types from the trace. This will inevitably lead to information loss, but the remaining trace provides enough substance for the various metrics.

3.0.1 General Properties

These properties are program wide and accumulated during the mining. The properties consist of the total amount of cases in the log ($|W|$) and the maximum encountered dependency value in the log ($\Rightarrow_W^{max}$).

3.0.2 Event

An event can be denoted as $< a >$ in trace notation.

For each event in the log two properties are stored, the frequency of the activity ($|a|$) and the maximum Dependency Value of the relations starting from this action ($a \Rightarrow_W^{max}$).
$\pi_{activity}(\varepsilon)$ returns the activity name of the event.
$\pi_{type}(\varepsilon)$ returns the activity type (either *start* or *finish*).

3.0.3 Pair Relation

A pair relation can be denoted as $< a, b >$ in trace notation or as $a \overline{>}_W b$ in a direct succession relation (see definition below)

Relation between two events in the log. Whenever a sequence of two events is encountered in a trace, a relation between these events is created. For each pair relation various properties are computed or tracked. These properties consist of the dependency value ($a \Rightarrow_W b$), long dependency value, parallel count($|a \,||_W b|$) , frequency ($|a \overline{>}_W b|$).

Definition 3.0.1. (Direct succession relation, $\overline{>}$) [5].

Let a and b be two interval activities (not instantaneous) in a log W, then

$$a \overline{>}_W b \text{ iff } \exists\, \sigma = \langle t_1, \ldots, t_n \rangle \text{ and } i \in \{2, \ldots, n-2\},\ j \in \{3, \ldots, n-1\}$$
$$\text{such that } \sigma \in W, t_i = a_{\text{finish}} \text{ and } t_j = b_{\text{start}} \text{ and}$$
$$\forall k \text{ such that } i < k < j \text{ we have that } \pi_{type}(t_k) \neq \text{start}$$

In the ordered trace $< a, b, d >$ every sequential pair is automatically in a direct succession relation: $(a \overline{>} b, b \overline{>} d)$

In the stamped trace $< a_{start}, b_{start}, b_{finish}, a_{finish}, d_{start}, d_{finish} >$ the relations consist of $(a \overline{>} d)$

Definition 3.0.2. (Parallelism relation, $||$) [5].

Let a and b be two interval activities (not instantaneous) in a log W, then

$$a \,||_W b \text{ iff } \exists\, \sigma = \langle t_1, \ldots, t_n \rangle \text{ and } i, j, u, v \in \{1, \ldots, n\}$$
$$\text{with } t_i = a_{\text{start}}, t_j = a_{\text{finish}} \text{ and } t_u = b_{\text{start}}, t_v = b_{\text{finish}}$$
$$\text{such that } u < i < v \ \vee\ i < u < j$$

In the trace $< a_{start}, b_{start}, b_{finish}, a_{finish}, d_{start}, d_{finish} >$ the events a and b are in a parallel relation ($a \,||_W b$).

Definition 3.0.3. (Dependency measure, $\Rightarrow_W$) [5]. Let a and b be two activities in a log W, then

$$a \Rightarrow_W b = \frac{|a \overline{>}_W b| + |b \overline{>}_W a|}{|a \overline{>}_W b| + |b \overline{>}_W a| + 2|a||_W b| + 1}$$

Loops of length one

A loop of length one can be denoted as $< a, a >$ in trace notation or as $a \overline{>}_W a$ in a direct succession relation.

Whenever a sequence consists of the same event the pair relation is flagged as a loop of length one.

Definition 3.0.4. (Loop of length one, $\Rightarrow_W$) [5]. Let a be a activity in a log W, then

$$a \Rightarrow_W a = \frac{|a >_W a|}{|a >_W a| + 1}$$

3.0.4 Triad Relation

A triad relation can be denoted as $< a, b, c >$ in trace notation or as $a \overline{>}_W b \overline{>}_W c$ in a direct succession relation

Relation between three sequential events in the log. This relation is created whenever a sequence of three events is encountered in a trace. The purpose of this relation is to extract length two loops.

Loops of length two

A loop of length two can be denoted as $< a, b, a >$ in trace notation or as $a \overline{>}_W b \overline{>}_W a$ in direct successino relation

Whenever a sequence consists of an event followed by another event, followed by the same event. The triad relation is flagged as a loop of length two.

Definition 3.0.5. (Loop of length two, $\Rightarrow^2_W$) [5]. Let a and b be activities in a log W, then

$$a \Rightarrow^2_W b = \frac{|a >^2_W b| + |b >^2_W a|}{|a >^2_W b| + |b >^2_W a| + 1}$$

3.0.5 Split Relation

The trace of a split relation can have the shape of $< a, b, ... >$ and $< a, c, ... >$ or $< a, b, c >$ and $< a, c, b >$. It can also be denoted in the following matter, $a \overline{<}_W b^{\wedge}_{\otimes} c$

Relation between three events in the log. Generated by combining a start event a with events b and c it is followed by. This triad of events represents a choice in the log, after a either b or c can happen or both in an unspecified order. This relation computes the split property which is then used to reason whether it is an AND or XOR split.

Definition 3.0.6. (Split relations, $\Rightarrow_W (\wedge)$) [5]. Let a, b and c be three activities in a log W, then

$$a \Rightarrow_W (b \wedge c) = \frac{|b \overline{>}_W c| + |c \overline{>}_W b| + 2|a||_W b|}{|a \overline{>}_W b| + |a \overline{>}_W c| + 1}$$

Long Relation

A long relation can be denoted as $< a, ..., b >$ in trace notation. It can also be denoted as $a >>>_W b$ (see definition below).

Relation between two events in the log. Whenever an activity is similar in frequency to another activity which it follows directly or indirectly, there is a possible indirect relation between the two. This relation is measured by using the indirect succession relation $a >>>_W b$ of events. Using this relation the long dependency measure ($\Rightarrow^l_W (\wedge)$) is computed.

Definition 3.0.7. (Long distance relation, $a >>>_W b$).

Let a and b be two interval activities in a log W, then
$a >>>_W b$ iff $\exists\, \sigma = \langle t_1, \ldots, t_n\rangle$ and $i \in \{1, \ldots, n-1\}$, $j \in \{2, \ldots, n\}$
such that $\sigma \in W, t_i = a$ and $t_j = b$ and
$\forall k$ such that $i < k < j$ we have that
$t_k \neq a$ and $t_k \neq b$

In other words, it computes the frequency of event a being (indirectly) followed by b. Where any event occurring between the pair can not be of the same activity as a or b.
In the ordered trace $< \underline{\mathbf{a}, \mathbf{b}}, d, a, c, c, \underline{\mathbf{a}, e, c, \mathbf{b}}, b, a, d >$ only the underlined parts count as a long distance relation for a and b.

A long distance dependency measure close to 1.0 indicates both that a is always followed by b and that the frequency of both activities in the log is about equal. Some long relations are already intrinsically represented in the WF-net, therefore an extra connection is not always necessary. A long relation between event a and event b should only be added if it is possible to go from a to the finish without visiting b.

Definition 3.0.8. (Long distance dependency measure, $\Rightarrow^l_W$) [14]. Let a and b be two activities in a log W, then

$$a \Rightarrow^l_W b = \left(\frac{2 \cdot (|a >>>_W b|)}{|a| + |b| + 1}\right) - \left(\frac{2 \cdot Abs(|a| - |b|)}{|a| + |b| + 1}\right)$$

Chapter 4

Research

4.1 Set-up of the application

The act of creating a model from the logging data requires a few steps or 'phases'. Different types of PM models require different numbers of phase. In section 4.1.3 we will describe which exact models are generated by our program, in this section we will go over each of these phases and explain how they work individually.
We start, of course, with loading the data. This is not really considered a phase in the application itself but is a necessary step in the complete process nonetheless. One of the powers of Process Mining lies in the fact that once a data set of suitable quality is provided, its algorithms can be applied to particular subsets of the data just as easily as the whole set. This enables analysis of the process on any desired granularity.
The first real step is the mining phase. In this phase we use mathematical formulas to derive relations between events from the log. The formulas are from the revised version of Andrea Burattins PhD thesis [5], which itself is a continuation of the works of Wil van der Aalst [8]. There are many types of relations to be found, some more obvious, and some more subtle. An example of a more obvious relation is the dependency relation between two events. This relation describes the probability of an event b happening after event a using a few different measures.
After the relations are mined from the log the conversion phase can begin. During this phase the mined data is used to construct one or more models. The complexity of this conversion from data to model highly depends on the type of model. The most complex type is the WF or Petri net, which require a lot of intermediate steps.

4.1.1 Loading the Data

As mentioned previously, the input data for any process mining application needs to adhere to a certain structure. The data needs to be formatted as a Log, a collection of Cases. Each case is either ordered chronological or provided with timestamps for the purpose of ordering it manually. With these basic requirements met, the program can be run. For the purpose of extracting meaningful insights it is often useful to provide more activity context to the data, exactly what type of context is up to the user of the program and the nature of the process. For a complete overview of the requirements and structure of the log data we refer you to Section 2.

4.1.2 Mining the Data

When we have successfully loaded the data into the program it is time to mine the data for its useful and hidden properties. For process mining this is all about relations. How do different events connect to each other? We use the formulas from Chapter 3 to extract the relations from Chapter 2. Besides these relations also some general statistics are recorded. The frequency of a log for example, whether the event is provided with a timestamp or the average execution time.

Mining the data is relatively simple and can be done in a single pass through over the log. For each case we iterate through the sequence and store each sequential pair as a pair relation and every sequential triad as a triad relation. For the long relations we keep track of the events already encountered in each single case and pair these with every new event encountered. While processing all these relations we flag any loops encountered.
After the cases are transformed we iterate over each relation and calculate the property values regarding each type of relation. For each type of relation there are both distinct and equal properties. The values and their formulas can be found in Chapter 3. The reason we do this step after creating the relations instead of combining these steps into a single pass is because we need a complete count of the frequency of each relation and event as input for some of the formulas.

After the Miner is finished we will be left with a collection of `Dictionaries` as data structure. Table 4.1 describes these `Dictionaries` with their respective Space Complexity (SC). Some remarks; the upper bound of a non WF-net log is only reachable when the log consists of the set of permutations with repetition of the set $\mathcal{A}$. In real life these kind of processes are very rare and above all, not relevant for process mining. There is no knowledge to be gained from applying process mining analysing techniques to these 'unstructured' logs since there is no driving process behind it.

By definition a WF-net has a *start* and *finish* event for which there are special properties, a *start* event has no input nodes and a *finish* event has no output nodes. This puts additional constraints on the Space Complexity.

Name	SC	SC WF-net
`event_frequencies`	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{A}\rvert$
`event_followed_by`	$\mathcal{O}(\lvert\mathcal{A}\rvert^2)$	$\mathcal{O}((\lvert\mathcal{A}\rvert-2)^2+2\cdot(\lvert\mathcal{A}\rvert-2))$
`pair_relations`	$\mathcal{O}(\lvert\mathcal{A}\rvert^2)$	$\mathcal{O}((\lvert\mathcal{A}\rvert-2)^2+2\cdot(\lvert\mathcal{A}\rvert-2))$
`triad_relations`	$\mathcal{O}(\lvert\mathcal{A}\rvert^3)$	$\mathcal{O}((\lvert\mathcal{A}\rvert-2)^3+2\cdot(\lvert\mathcal{A}\rvert-2)^2)$
`split_relations`	$\mathcal{O}(\lvert\mathcal{A}\rvert\cdot C(\lvert\mathcal{A}\rvert,2))$	$\mathcal{O}((\lvert\mathcal{A}\rvert-1)\cdot C(\lvert\mathcal{A}\rvert-1,2))$

Table 4.1: Datastructure of the Miner with its Space Complexities (SC). $\mathcal{A}$ is the set of activities contained in the log W. SC is the complexity for any log, SC WF-net is the complexity for a WF-net log.

For calculating the SC of the pair and triad relations of a regular log we compute a standard permutation with repetition (n^k) where the n is the number of events ($|\mathcal{A}|$) and k is the relation size (in our case either 2 or 3). A relation is ultimately a combination of two or three events, where an event can follow itself. For the WF-net log we use the following reasoning for the SC computation. If we consider the size of a relation to be n. We first 'remove' the start and finish event from the list of events for permutation calculation $(|\mathcal{A}|-2)^k$. Next we add the possibilities where a start event is appended to the start or a finish event is appended to the end of a relation. For a relation of size k we can append to the relations of size $k-1$ to reach the desired size. The SC of the `event_followed_by` table is the same size as the `pair_relations` table. For any trace $< a, b >$ both a pair relation for the combination of a and b and an entry that b follows a will be created.
In a regular log a split involves three events. A split originates from any event and is tailed by any two events, including the origin, but not twice the same. Therefore we multiply the number of events by the number of two sized combinations which can be formed from the group of events. The WF-net log can't contain a split originating from the finish event and can't contain the start event in one of its split end points. Hence we subtract one from the number of events for its SC.

4.1.3 Converting the Data

After mining all the relations from the log we can start with constructing the models. The program currently supports three kind of models, a **WF-net**, **Flow Graph** and **Dotted Chart**. All these models provide different views into the process so they can be combined to provide deeper analytical insights. By far the most challenging model to convert to is the WF-net, with the flow graph coming in second. The dotted chart boils down to a specific way of plotting the data and needs no substantial conversion phase.

For the **Dotted Chart** a chart is rendered from the log. For every event a dot is plotted at a position corresponding to its timestamp on the x-axis and case code on the y-axis [10]. Every dot is then color coded according to its event tag and every case is ordered on its starting timestamp. The resulting chart give a intuitive overview of the shape of the data and is especially useful for inspecting the suitability of the log data before proceeding with any more advanced process mining models. Three renderings of a dotted chart can be found in Chapter 6.

The **Flow Graph** uses the pair relations in the `pair_relations` from the mining phase to construct a graph similar to a flow network. In this graph each node represents an event and each edge has a capacity corresponding to how often this relation occurs in the log. This type of model is useful for discovering the broad lines of a process; edges with a value below a certain threshold can be ignored to provide an overview of the core flow. An example of a flow graph model can be found in Figure 6.6

The **WF-net** will be discussed in more detail since the complexity of this conversion demands it and because this is where the focus of this thesis lies. Converting to a WF-net is a step by step process. For this process a number of thresholds can be set which influence the resulting model. Here we list the thresholds and present, in a brief description, the ratio of the specific threshold [5].

Relative To Best Threshold (RTB) This parameter indicates that we are going to accept the current edge (i.e., to insert the edge into the resulting control-flow network) if the difference between the value of the dependency measure computed for it and the greatest value of the dependency measure computed over all the edges is lower than the value of this parameter.
Positive Observations Threshold (PO) With this parameter we can control the minimum number of times that a dependency relation must be observed between two activities: the relation is considered only when this number is above the parameter value.
Dependency Threshold (D) This parameter is useful to discard all the relations whose dependency measure is below the value of the parameter.
Length-one Loop Threshold (LOL) This parameter indicates that we are going to insert a length-one loop (i.e., a self loop) only if the corresponding measure is above the value of this parameter.
Length-two Loop Threshold (LTL) This parameter indicates that we are going to insert a length-two loop only if the corresponding measure is above the value of this parameter.
Long Distance Threshold (LD) With this parameter we can control the minimum value of the long distance measure in order to insert the dependency into the final model
AND Threshold (A) This parameter is used to distinguish between AND

and XOR splits (if there is more than one connection exiting from an activity): if the AND measure is above (or equal) to the threshold, an AND-split is introduced, otherwise a XOR-split is introduced.

All these parameters have values between -1 and 1, except for the **Positive Observations Threshold** which requires a natural number as input. For every threshold there is a default value, but to acquire the best results the thresholds need to be tweaked. By choosing the right thresholds the 'roughness' of the model can also be adjusted. With high values for all the thresholds (except the AND threshold) the result will be a more robust model with only the most significant process flows. In the opposite case the model will be more precise and inclusive, but also substantially more error prone and incomprehensible. It is up the user to find the right value for each parameter to generate the most suitable model for his needs.

These thresholds serve as parameters for the conversion phase whose steps are described below. Further explanation for the steps which require it can be found after the list.

1. Create a transition for each activity in the log.
2. For each pair relation from `pair_relations` which satisfies the thresholds (RTB, PO, D), connect its transitions with a place.
3. Implement each split from `split_relations` according to the thresholds (A) and the connections in the net.
4. Add the long distance relations into the net if they satisfy the threshold (LD) and requirements.
5. If the last step created new splits, implement those into the net.
6. Join the net for each transition with multiple incoming connections.
7. Add loops to the net if they satisfy the loop thresholds (LOL, LTL).

(8.) (*optional*) Clean the net.

In order to make a sound and accurate model of the log it is occasionally necessary to add additional *hidden* transitions to the net after step 1. These transitions exists to enforce the structure and schemantics of a WF-net. Most often it is needed when layering multiple splits in step 3,5 or joining multiple splits in step 7.

We define three types of transitions in a WF-net.
The **Activity Transition** represents the execution of an activity, there is only one transition for every activity in the log W.

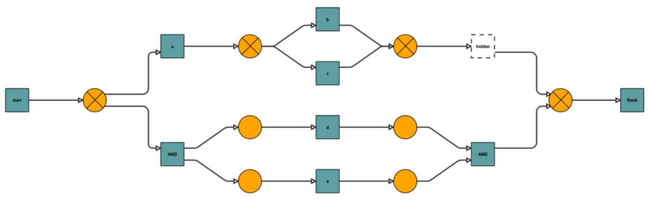


Figure 4.1: WF-net with a Hidden and AND transition. The hidden transition is denoted by the dotted lines, the AND transition can be recognized by its label on the transition: 'AND'. Both transitions behave the same as a normal activity transition, except for the fact that they are silent transitions and thus do not appear in a trace.

The purpose of a **Hidden Transition** is to keep the structure of a WF-net intact and is added in specific situations. One of these situations is for example after a XOR-join.
An **AND Transition** is, if needed, added when an AND split is nested in a XOR split or as part of an AND join.

Step 1 & 2: Connecting the transitions

Two transitions are connected with a place when

For any pair relation $a \,\overline{>}_W b$,

$$a \Rightarrow_W b \geq D \;\wedge\; (\Rightarrow_W^{max} - a \Rightarrow_W b) \leq RTB \;\wedge\; |a \overline{>}_W b| \geq PO$$

If these conditions are satisfied then we write $(a \,\overline{>}_W b)^\top$, meaning the transition is added into the net. In the contrary case we write $(a \,\overline{>}_W b)^\perp$

Step 3: Splitting the net

After all the events from the log are added as transitions and these transitions are linked according to the pair relations, it is time to split the net. For this step there are two methods, both with their own purpose. Firstly, a naive method which is suitable for constructing simple models in a fast manner. Secondly, a method which uses a set structure to correctly compute more complex models. These methods are the **Naive Split Method** and **SplitSet Method**, respectively.

The starting structure of both methods is the same, they begin with iterating through the mined split relations from the `split_relations` Dictionary. A split is viable to implement into the net when

For any split relation $a \overline{<}_W b^{\wedge}_{\otimes}c$,
$a \neq b \wedge a \neq c \ \wedge \ (a \Rightarrow_W b)^{\top} \wedge (a \Rightarrow_W c)^{\top}$

If these conditions are satisfied it depends on the value of A whether the split is an AND or XOR split. If $a \Rightarrow_W (b \wedge c) < A$ then it is a XOR split, else an AND split.

$$a \overline{<}_W b^{\wedge}_{\otimes}c = \begin{cases} \wedge \text{ (AND)} & \text{if } a \Rightarrow_W (b \wedge c) \geq A \\ \otimes \text{ (XOR)} & \text{if } a \Rightarrow_W (b \wedge c) < A \end{cases}$$

When the split type is determined we can change the notation of a split. $a \overline{<}_W b{\otimes}c$ denotes a XOR split between b and c and $a \overline{<}_W b{\wedge}c$ an AND split.

After iterating through all the split relations and determining the type of split it is time to implement them into the net. This is where the two methods diverge. First we discuss the **Naive Split Method**.
We again iterate through the splits. For every transition encountered we check the number of splits starting from this point. Here we distinguish two cases: there is one split; there are multiple splits. Whenever there is a single split we can implement it straightforward. We will illustrate how this works with a simple Petri net.

N = ({a,b,c}, {$p1$,$p2$}, {(a,$p1$), (a,$p2$), ($p1$,b), ($p2$,c)})

To transform this into an AND split with source a and children b,c we don't need to do anything (step 2. already created the AND template for us). When the single split is a XOR we bundle the two places into one, thus creating the XOR template.

$N^{AND} = N$
N^{XOR} = ({a,b,c}, {$p1$}, {(a,$p1$), ($p1$,b), ($p1$,c)})

It gets more complicated when there are multiple splits originating from the same point however. These splits have to be combined in order to create a proper WF-net. Here we again distinguish two possibilities: all the splits are of the same type; the splits are of mixed types.
When the splits are all equal in type we simply add all distinct activities together into one big split. If we have the following collection $\{a \overline{<}_W b \otimes c, a \overline{<}_W b \otimes d\}$ we can combine this into the single XOR $a \overline{<}_W b \otimes c \otimes d$. Note that the duplicate b translates to a single entry in the XOR. Figure 4.3 shows the Petri net implementation of this XOR. An collection of multiple AND splits can be implemented with the same process.
In case of mixed types we need to chain the splits to create nested splits in the net. We will illustrate how this works with a Petri net and a collection

of splits originating from transition a.

N = ({a,b,c,d}, {$p1$,$p2$,$p3$}, {(a,$p1$), (a,$p2$), (a,$p3$), ($p1$,b), ($p2$,c), ($p3$,d)})
$\{a \overline{<}_W b \otimes c, a \overline{<}_W b \wedge d, a \overline{<}_W c \otimes d\}$

First we order the collection on the split type, with XOR before AND. The reason we do is to decrease the complexity of the resulting net, which additionally increases the readability. Figure 4.2 illustrates two Petri nets which are equivalent in execution, but different in structure. We see that version b) is more straightforward in interpretation. After ordering the collection looks like $\{a \overline{<}_W b \otimes c, a \overline{<}_W c \otimes d, a \overline{<}_W b \wedge d\}$.
Now we take all the XOR splits and implement these in the same fashion as a multiple XOR split. This results in the following net:
N = ({a,b,c,d}, {$p1$}, {(a,$p1$), ($p1$,b), ($p1$,c), ($p1$,d)})
$\{a \overline{<}_W b \wedge d\}$

Now only the AND split(s) are left. The next step is to bundle the activities in an AND split together inside of the XOR. A new AND transition will be added between the XOR place and the transitions in the split.

N = ({a,b,c,d,AND}, {$p1$,$p2$,$p3$}, {(a,$p1$), ($p1$,AND), ($p1$,c), (AND,$p2$), (AND,$p3$), ($p2$,b), ($p2$,d)})

With this our merged split is complete according to the **Naive Split Method**. This method has a drawbacks however. The 'depth' of a split can only be two: a XOR followed by an AND. This greatly limits the complexity of a model and thereby also how well the generated model represents the underlying process. The **SplitSet Method** is therefore a better option for more complicated processes. This method translates the set of splits originating from an activity to a collection of *splitsetS*. A *splitset* is a type of set which was introduced for the very purpose of process mining. These splitsets are then added together until they form one single splitset, this union of splitsets determines the shape of the split in the net. The scemantics are straigforward: square braces are used to denote a splitset. A set can contain two types of elements: an activity or another splitset. The shape of the splitset determines the relation between the activities it contains. [a] is a singleton splitset containing the activity a. [a,b] is a splitset containing activity a and b, because both activities share the same set we can say they are in a XOR relation. The set [[a],[b]] contains the same activities but in an AND relation because they are divided into different subsets.
As was said before, this method supports more complex processes, but it comes at a price. The addition of splitsets is not commutative which means the order in which they are added matters. In order to 'discover' a working order we need to try every combination, which means that in the worst case

scenario the whole set of permutations needs to be checked. However, some measures are in place to combat this problem and to ensure that the probability of going through the whole permutation set is minimal.

The **SplitSet Method** has the structure of a loop, first the splitsets are merged into one according to one of the permutation possibilities. This union splitset is then checked on its performance qualities, in other words, how well it reflects the behaviour encountered in the log. This quality check results in a number in the range of 0 to 1 which correlation to a percentage. When the quality value is 1 the splitset perfectly fits the behaviour we want to capture and we can stop looking for alternatives. If it is any number beneath 1 we only save the splitset if its the highest number encountered up to that point and we continue looking for other alternative implementations. It can happen that no perfect splitset can be found as sometimes the log is just too unstructured and contains behaviour which can not be modelled using a WF nets execution properties. In such a case after iterating through every possibility we simply take the best encountered splitset.

In order to evaluate all the possible unions splitsets a small excerpt of the log is extracted. A number of traces from the log, the exact amount can be set manually as a parameter, is inspected and from these traces relevant subtraces are extracted. A subtrace is relevant if it starts with the origin of the split event and includes all the subsequent events which are part of the union splitset. So if we have a splitset with origin event a of the shape $[[b, c], d]$ and a trace of the shape $< start, \mathbf{a}, \mathbf{d}, \mathbf{b}, e, b, finish >$ only the bold part is relevant and will be added to the subtrace. Next the union splitset is converted to a tiny Petri Net using the semantics as described above (XOR for same set elements, AND for different set elements). The collection of subtraces is then executed on this Petri Net and the percentage of successful executions is then used to assess the quality of the splitset and mapped to a value in the range of 0 to 1.

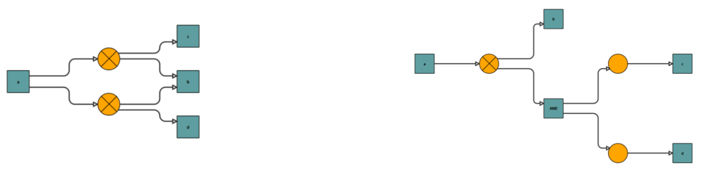
Step 5: Long distance relations

As a reminder, a long distance relation indicate cases where an event b depends indirectly on another event a to be executed.[14] A connection should only be added if there is no path from a to the finish without visiting b.

For any long relation $a >>>_W b$,

$$a \Rightarrow^l_W b \geq LD \vee \text{the } \textit{finish} \text{ can be reached from } a \text{ without encountering } b$$

If these conditions are satisfied a connection between a and b will be introduced into the net. This will be done by introducing a place between



(a) Poor readability (b) Better Readability

Figure 4.2: Two Petri nets differing in readability but equal in execution. At first sight of 4.2a the three activities seem to share the same 'level' of splits, upon closer inspection we see however that activity *b* is in a XOR relation with *c* and *d* and can thus never appear in the same trace. 4.2b shows this behaviour more effectively by 'chaining' the splits, and is thereby the preferred option.

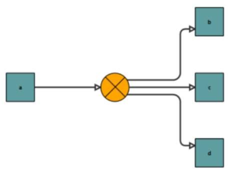


Figure 4.3: Multiple XOR split. From activity *a* either *b*,*c* or *d* can be fired. There is no limit to the amount of places connected to the same XOR place to create a multiple XOR split.

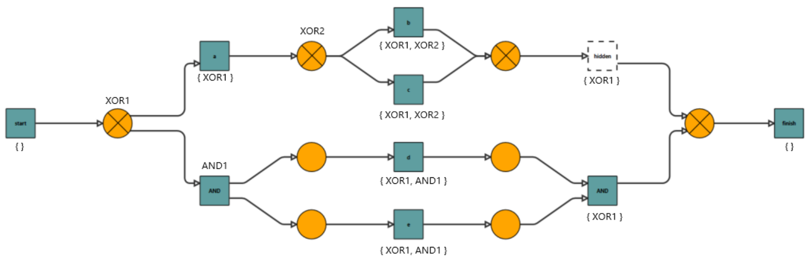


Figure 4.4: A WF-net with for each transition its split history displayed. The labels above a node show the name of the node, the list underneath a transition show the history of that transition, in the order that the splits are encountered.

the activities, as a result the net changed its structure and the split and join steps have to be repeated.

Step 6: Joining the net

After each split relation has been added to the net it is time for the next step, joining these splits. The intuition behind this step is that whenever a split occurs, a join must follow after. Therefore the approach to finding joins it to keep track of the splits which occurred before every activity, the *split history* (H) so to speak. This split history consists of a `Dictionary` datatype where each transition in the net is mapped to a list of the splits occurring before it. In order to build this dataset the net is traversed from *start* to *finish* where a list with the encountered splits is passed on between each transition. When this transition is flagged as a split, it appends a tuple with the type and id of the split to the list. Initially, at the start transition, the list is empty. In figure 4.4 you can see a visualization of what the split history looks like for each transition.

$H[a] = \{(xor, 1), (and, 2)\}$

After constructing the split history we start with the main loop. We iterate through the transitions until we find a transition t with more than one connection as input. Now we search back through the net and gather a collection of transitions T_{parent} which are connected to t through a place. In other words, $T_{parent} = \bullet \bullet t$. From T_{parent} we select the transitions which have the longest split history and for which the last element in their split history list is equal. For the net in Figure 4.4 this would be transition ?? and ??. By selecting only the longest split history we ensure we don't close any 'big' splits which have unclosed splits nested in them. Doing that will

unable these splits to join !EXPLAIN!. Implementing a join in the net is very similar to implementing a split. For an AND join no alteration of the net is needed. For a XOR join the places are merged into one and a hidden transition is added to connect this place to *t*. This hidden transition ensures that the net can be joined further when splits are chained/layered. Such a hidden transition is not always necessary and can be removed in the optional step 8. to improve the readability of the graph.
After the net is adjusted we repeat the same process again, the net has changed and there might be new joins available which weren't available before. So we traverse the tree again to build the split history and likewise iterate through each transition. After a complete pass has been done through each transition and no join was discovered, we consider this step done.

Step 7.

This step is very straightforward. All the loops of length one and two are stored during the mining phase so we only need to check if they meet the thresholds requirements.

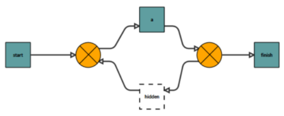
For any length one loop $a \overline{>}_W a$,
$a \Rightarrow_W a \geq \text{LOL}$

For any length two loop $a \overline{>}_W b \overline{>}_W a$,
$a \Rightarrow^2_W b \geq \text{LTL}$

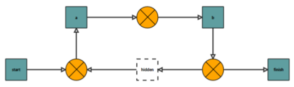
If a loop is accepted we need to implement this looping behaviour into the semantics and execution of a WF-net. Templates for the loops are shown in Figure 4.5. The construction is as follows. A new hidden transition is added to the net and connected to the input place(s) and output place(s) of the loop. For a loop of length one this means the input and output of *a*, for loops of length two its the input of *a* and output of *b*. If there are multiple input or output places the connections are joined and split in a XOR fashion.

4.2 Limitations of the Heuristics Miner algorithm

When trying to construct a model from logging data the question is often "how close can we get to a suitable model" instead of "how can we get the *best* suitable model". Building a model from a series of chronologically ordered events is difficult by definition. Even with a noise-free data set there is still a significant chance that the mined model does not represent the data perfectly. The Heuristics Miner is no exception, in the section below we will discuss some of the limitations we encountered when implementing the algorithm. In section 4.3 we will discuss any improvements we made to the algorithm to combat these limitations.



(a) Loop of length one



(b) Loop of lengh two

Figure 4.5: Looping templates in WF-net notation. A loop of length one enables a single transition to be repeated infinitely, a loop of lenght two enables an infinite sequence of alternating transitions.

4.2.1 No differentiation between 'skips' and 'splits'

In a process it can happen that sometimes a certain activity or series of activities is skipped. This could be because there is an optional part in the process, or because different types of resources need different types of treatments. Lets for example consider the process of recycling a piece of paper and a cardboard box. When placing a cardboard box in the bin the box needs to be flattened, otherwise it takes up to much space. Paper is however inherently already flat, and this step can thus be skipped.
Skipping an event in a process occurs quite regularly, the Heuristics Miner does not recognise such an event however. A simple log with two cases can illustrate why this is a problem.

$$< start, a, b, finish >$$
$$< start, a, finish >$$

When we work out the formula for AND \ XOR relations it detects an AND relation for parent a and its children b and $finish$ (with the standard threshold value of 0.1).

$$\begin{aligned} a \Rightarrow_W (b \wedge finish) &= \frac{|b \overline{>}_W finish| + |finish \overline{>}_W b| + 2|a||_W b|}{|a \overline{>}_W b| + |a \overline{>}_W finish| + 1} \\ &= \frac{1 + 0 + 2 \cdot 0}{1 + 1 + 1} \\ &= \frac{1}{3} \end{aligned}$$

With this log as input and the standard value for the AND Threshold, the Heuristics Miner will thus produce the model in Figure 4.6 When this petrinet is executed it produces $< start, a, b, finish >$ as only output, the $< start, a, finish >$ trace can not be replicated with this model because for activity $finish$ to fire, b is required to fire prior.

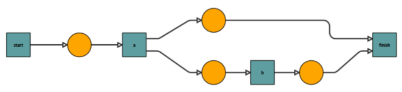


Figure 4.6: Faulty WF-net generated by the Heuristis Miner. The net should be able to parse the trace $< a, finish >$ but after firing transition a the transition *finish* is not enabled since the outgoing place of b does not hold a token.

4.3 Improvements of the Heuristics Miner

This section is devoted to discussing possible improvements to the Heuristics Miner. These improvements where either actually implemented in the program or simply reasoned about. The number of every subsection references to the number of the subsection in section 4.2

4.3.1 Implementing 'skips' in the net

Detecting a split of length one (skips one activity) is quite a trivial task. The formula for the split relation (Definition 3.0.6) can be used for this purpose. We can define this as follows:

For any activities a, b and c in a split relation, there exist a length one skip when either

$$|b \overline{>}_W c| = 0 \;\wedge\; |b \overline{>}_W c| \neq 0 \text{ or}$$
$$|b \overline{>}_W c| \neq 0 \;\wedge\; |b \overline{>}_W c| = 0$$

More intuitively, this means that whenever an activity a has two children b, c and one of the children follows the other, but not vice versa, we can say that there exist a length one skip.

Chapter 5

Related Work

As one of the founders of the discipline of process mining itself, Wil van der Aalst provides a solid knowledge foundation with his book *Process Mining: Data Science in action* [8]. It covers a wild variety of topics including a primitive version of the Heuristic Miner.

Process Mining Techniques in Business Environments [5] improves upon this work by making use of timestamps to better determine whether activities take place in succession or in parallel. Andrea Brattin dubbed this version of the algorithm the Heuristic Miner+.

Sofie De Cnudde, Jan Claes and Geert Poels [7] analysed the shortcomings of the HM and introduced more accurate measured for defining loops.

W.J.M.M Weijters and J.T.S. Ribeiro revised the HM with the *Flexible heuristics Miner* [14]. This updated version tries to exploit all the advantages of the basic ideas underlying the Heurstic Miner. It, among other things, improves the required conditions for detecting long relations and proposes a new approach to the characterization of splits and joins.

Chapter 6

Results

In this chapter we will explain the shape of the logging data used during development and each of the generated models generated from it.

The input data is gathered from a manufacturing hall specialized in producing custom tailored car mats for various car manufactures. The process starts off with a big industrial sized sheet of car mat. This sheet is than subsequently cut into the right shapes, reinforced, embroidered, packaged, inspected, sealed and finally send of to the client. It is in no way a linear process however, every order is custom and therefore needs a unique arrangement of processing. Additional to that, occasional errors occur which need to be corrected, which can lead to steps being repeated or other diversions in the process path. The semi organized structure of the process makes it a suitable data set to test both the strengths and limits of Process Mining.

A small excerpt of the data can be found in table 6.1. Every **ActualOperationId** is a unique identifier for that row and thus maps to an activity in the process. A **POCO** can be seen as a case, it describes a bundle of orders which undergo the same processing. The next three columns indicate when the activity took place in the process. The **Started** column contains a timestamp of when the activity commenced, the **FinishedAtualOperation** timestamp is added when the activity is completed. As we can see from the table the logging data is not complete, **NULL** values can be present in the last column. When this happens the Miner takes the activity to be instantaneous and duplicates the **Started** value to the **NULL** value.

In order to create a visual understanding of the manufacturing process we rendered a simple WF-net with only the pair relations, which is thus similar to a flow network, using the Heuristic Miner (Figure 6.1). An added bonus to using this method is that the HM additionally filters out noise and irrelevant information. What is considered irrelevant depends on the

ActualOperationId	POCO	Activity	Started	FinishedActualOperation
542284	I>002/708151	Snijden	2021-09-02 12:43:17.480	2021-09-02 13:48:44.310
545003	I>002/708151	Sealen	2021-09-07 10:21:12.950	NULL
545159	I>002/708151	Sealen	2021-09-07 11:35:34.687	2021-09-07 11:37:18.393
545159	I>002/708151	Sealen	2021-09-07 11:35:34.687	2021-09-07 11:37:22.660
558634	I>002/715203	Snijden	2021-09-27 18:45:05.060	2021-09-27 19:03:32.383

Table 6.1: Small excerpt from the dataset

parameters. The WF Net in Figure 6.1 was then adjusted by the domain expert to better fit his perception of the process.

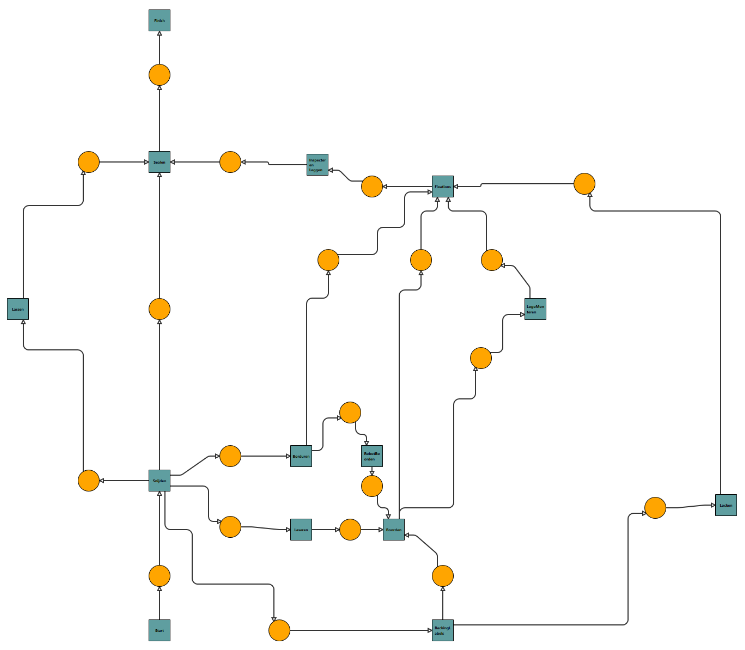


Figure 6.1: Mock WF-Net generated only from the mined pair relations (without the split or join steps) and afterwards adjusted by the domain expert by removing invalid relations. This mock net serves as a guideline for what the flow of the actual generated model should resemble.

In the remainder of this section we will show the rendered models starting with the **Dotted Chart** and then the **Flow Network** and **WF-Net**, respectively.

To give a rough impression of the log a dotted chart is often rendered. For large time periods such as months, displayed in Figure 6.2, the dotted chart enables the human viewer to quickly identify patterns which are harder to see from browsing the log in tabular format [10]. When zooming in on

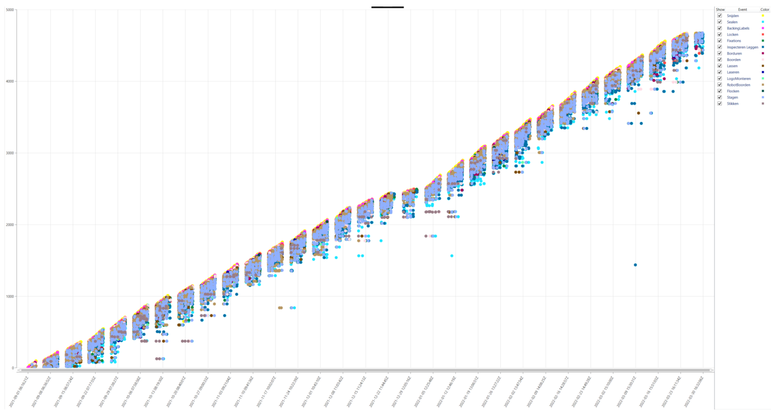


Figure 6.2: Dotted Chart displaying the data of three months from 01-09-2021 to 30-03-2022. Every dot represents the start of an activity and its color corresponds to the type of activity. Legend in Figure 6.5

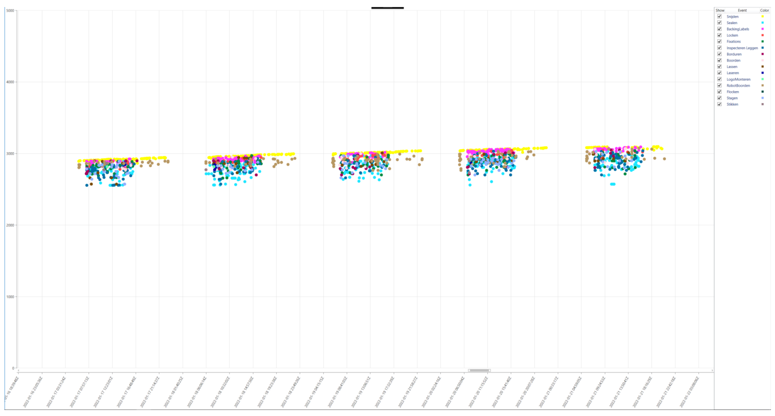


Figure 6.3: Dotted Chart displaying a one week time period from 16-01-2022 to 22-01-2022. Every dot represents the start of an activity and its color corresponds to the type of activity. Legend in Figure 6.5

smaller time periods such as days, see Figure 6.4 and Figure 6.3, the shape of a business day or week can be observed. A good way to read the chart at a point in time is from top to bottom. For any point on the x axis the top events are the start of a case. The lower the dot is charted the later it occurs in a case. From Figure 6.3 we can for example see that (most) cases begin with the event 'Snijden' because the bulk of the top dots are yellow.

A flow network is a great tool to discover the connectivity between events and additionally uncover the main flow of the process. Figure 6.6 shows such a flow network of the data. Every event is represented as an orange square. Every weighted edge represents a step in the process or in other words, that the target node occurs after the source node in the log. The higher the

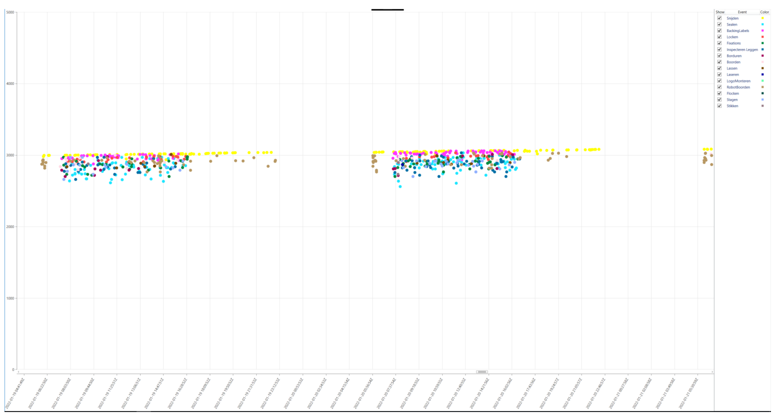
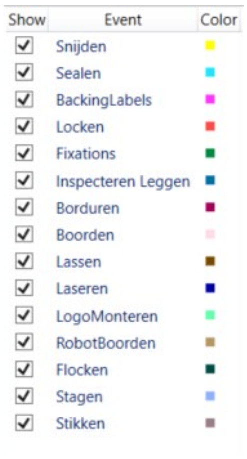


Figure 6.4: Dotted Chart displaying a two days time period from 19-01-2022 to 21-01-2022. Every dot represents the start of an activity and its color corresponds to the type of activity. Legend in Figure 6.5



Show	Event	Color
✓	Snijden	🟨
✓	Sealen	🟦
✓	BackingLabels	🟪
✓	Locken	🟥
✓	Fixations	🟩
✓	Inspecteren Leggen	🟦
✓	Borduren	🟪
✓	Boorden	⬜
✓	Lassen	🟫
✓	Laseren	🟦
✓	LogoMonteren	🟩
✓	RobotBoorden	🟫
✓	Flocken	🟩
✓	Stagen	🟦
✓	Stikken	🟫

Figure 6.5: Zoomed in legend of the Dotted Chart graphs, applicable to Figure 6.2, 6.3 and 6.4

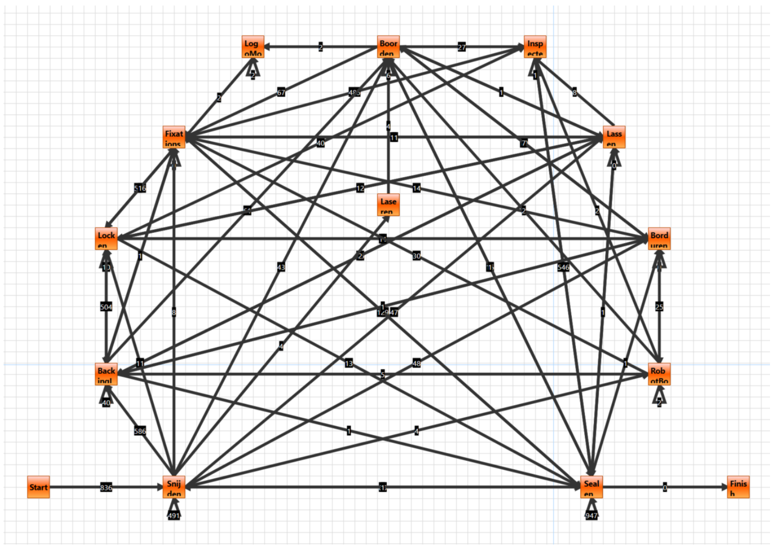


Figure 6.6: Flow Network generated from the logging data. The frequency of each relation is displayed on the corresponding edge. We observe a high connectivity between each individual activity in the log.

Relative To Best	0.1
Positive Observations	10
Dependency	0.1
Length One Loop	0.9
Length Two Loop	0.9
Long Distance	0.9
Split	0.1

Table 6.2: Default parameter settings for the Heuristic Miner [5]

weight of an edge, the more that step occurs in the log. Filtering out edges with relatively low weight will enable the viewer to see the primary connectivity between events in the log.

A WF-Net doesn't only model the structure of the process but also its behaviour. It is therefore the most complex model in terms of both structure and schematics. The WF-Net displayed in Figure 6.7 is rendered with the 'standard' parameter settings for the Heuristic Miner, found in Table 6.2 [5].

Unfortunately this net does not model the reality particularly well. When we run the log through the net we get a Parsing Measure of 0 and a Partial Parsing Measure of approximately 0.286. This means that only well under a third of all events in the traces is parsed correctly. If we inspect the net visually there are some errors observable in the structure, the main one be-

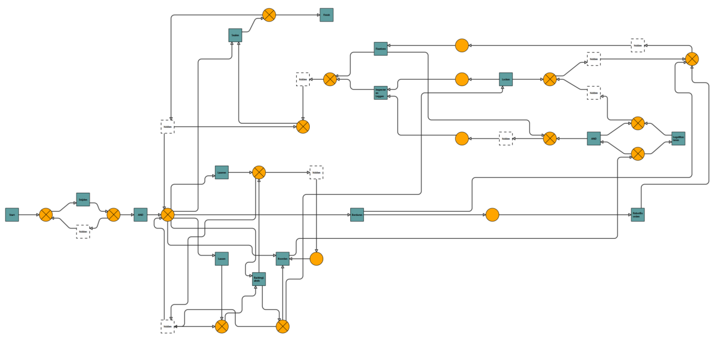


Figure 6.7: WF-Net generated from the logging data with the following parameters: { Relative To Best=0.1, Minimal Positive Observations=10, Dependency=0.1, Length One Loop=0.9. Length Two Loop=0.9, Long Distance=0.9, Split=0.1 }

ing the general lack of tokens which prevent a lot of transitions from firing when they are prompted during trace parsing.

Let us do an example run through the net to illustrate its issues, we fire the start transition and place a token at the first place. From this place we can fire the 'Snijden' transition, this transition can be repeated because the hidden transition below it forms a loop one. After we fire the transition a token is places on what we can call the 'crossroad place' which consists of a total of 6 outgoing edges and 3 incoming. From here we encounter the problem which the net faces: most of the transitions require more than one enabled input place which can't occur with a single token in the net. The transitions which are enabled are 'Laseren', 'Borduren' and 'Lassen'. Firing 'Laseren' or 'Lassen' leads to a deadlock. We can fire 'Borduren' and subsequently 'RobotBorden' which leaves us with two tokens in the top right XOR place. From here we are left with firing 'Fixations' along with some hidden transitions, which leads us inevitably to our final deadlock. A big portion of the net can thus never be reached, including the finish transition which automatically disables any trance from being completely parsed.

Chapter 7

Conclusions

The goal of this thesis was to create a set of intuitive and easy to use process mining tools for the business user. For this purpose a program was created to generate three different kinds of models from logging data. The **Dotted Chart**, **Flow Network** and **WF-net**. Each of these models serves a distinct purpose and provides a different view of the logging data. The **Dotted Chart** gives a quick overview of the shape of the data and shows patterns which are invisible in tabular format. The **Flow Network** shows the connectivity between events in the log and exposes the main flow of the process. The **WF-net** describes the flow and behaviour of the log and can be used to get a thorough understanding of the actual process.

The performance of the first two models is on par and provides the knowledge which is described above, because of their straightforward nature implementing these was also forthright. The performance of the last model, the **WF-net**, leaves room for improvement however. The fitness on the real live dataset is poor (Partial Parsing Measure of 0.286) and when visually inspecting the network it is clear that there are intrinsic faults present, including deadlocks. These problems can be attributed to two parts of the process: the method of model generation and the quality of the dataset.

First we consider the quality of the dataset. When inspecting the flow model we see a highly convoluted network with a high number of input and output edges per node. The dotted chart also shows a irregular mixture of colours in the bottom part of each daily cluster. Both these findings point to a disorderly process. It is difficult to translate a process to a WF-net with as only source the logging data if the process is relatively unstructured.

Secondly, we evaluate the method used. Despite the popularity of the Heuristic Miner in research and practise [7] it is not a fail proof algorithm. Detailed descriptions of the algorithm which seemed most useful when implementing the algorithm were often outdated. In the meantime there are more advanced miners developed (Inductive Miner [4], Fuzzy Miner [15]) which often offer better performance, especially when faced with unstructured processes.

Despite not being able to generate a perfect representation of the process the Heuristic Miner can still be used for other purposes. When tweaking the parameters it is a great tool to discover central and insignificant flows in the process, or to generate a top level WF-net with only the core events. Even when creating a 'faulty' model the program still serves up to part of its purpose, it forces the business user to think about the actual process an if and how it should be improved.

Bibliography

[1] W.M.P. Aalst, van der. The application of petri-nets to workflow management. *Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[2] W.M.P. Aalst, van der, K.M. Hee, van, A.H.M. Hofstede, ter, N. Sidorova, H.M.W. Verbeek, M. Voorhoeve, and M.T. Wynn. Soundness of workflow nets : classification, decidability, and analysis. *Formal Aspects of Computing*, 23(3):333–363, 2011.

[3] Daniel Beverungen. Exploring the interplay of the design and emergence of business processes as organizational routines. *Business Information Systems Engineering*, 6(4):191–202, 2014.

[4] Romero C. Bogarín A, Cerezo R. Discovering learning processes using inductive miner: A case study with learning management systems (lmss). *Psicothema*, 30(3):322–329, 2018.

[5] Andrea Burattin. *Process Mining Techniques in Business Environments: Theoretical Aspects, Algorithms, Techniques and Open Challenges in Process Mining.* Springer Publishing Company, Incorporated, 2015.

[6] Manuel Castells. *The Information Age: Economy, Society and Culture Vol. I*, volume 1. Blackwell Publishers, 2010.

[7] Sofie De Cnudde, Jan Claes, and Geert Poels. Improving the quality of the heuristics miner in prom 6.2. *Expert Systems with Applications*, 41(17):7678–7690, 2014.

[8] Van der Aalst W. *Process Mining: Data Science in Action.*

[9] Dhruba Karki. Can you guess how much data is generated every day?, 8 2020.

[10] Thomas Molka, Wasif Gilani, and Xiao-Jun Zeng. Dotted chart and control-flow analysis for a loan application process. In Marcello La Rosa and Pnina Soffer, editors, *Business Process Management Workshops*, pages 223–224, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[11] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[12] Wolfgang Reisig and Grzegorz Rozenberg. *Lectures on Petri Nets I: Basic Models*, volume 1. Springer Publishing Company, Incorporated, 1988.

[13] Wil Van Der Aalst. Process mining. *Commun. ACM*, 55(8):76–83, August 2012.

[14] A.J.M.M. Weijters and J.T.S. Ribeiro. Flexible heuristics miner (fhm). In *2011 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 310–317, 2011.

[15] Danica Virlianda Marsha Yutika Amelia Effendi, Riyanarto Sarno. Improved fuzzy miner algorithm for business process discovery. *telkomnika*, 19(6), 2021.